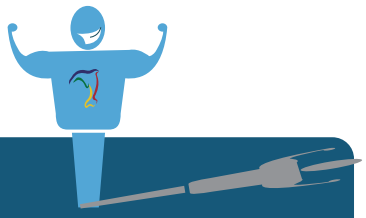
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

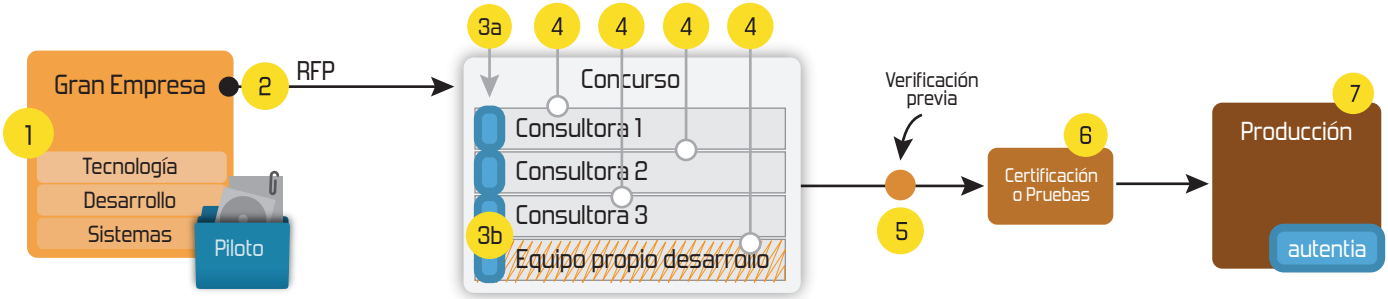
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Tutorial desarrollado por

Cristóbal González Almirón

Consultor de desarrollo de proyectos informáticos. Su experiencia profesional se ha desarrollado en empresas como Compaq, HP, Mapfre, Endesa, Repsol, Universidad Autónoma de Madrid, en las áreas de Desarrollo de Software (Orientado a Objetos), tecnologías de Internet, Técnica de Sistemas de alta disponibilidad y formación a usuarios.

Contacte con Cristóbal González
criskerberos-tutoriales@yahoo.com

Descargar este documento en formato PDF DependencyInjector.pdf

Fecha de creación del tutorial: 2008-01-03

Patrón de Inyección de dependencias

Patrón de Inyección de dependencias
Introducción
Diseños tradicionales
Diseñando nuestro patrón de inyección de dependencias DependencyInyector
Esquema de funcionamiento
Ejemplo de implementación en Java 5
- Init.java
- InstanceA.java
- InstanceB.jave
- DependencyInjector.java
- InstanceConfigurationReader.java
- InstanceConfiguration.java
- ParameterConfiguration.java
- InstanceCrearor.java
- InstanceConfigurator.java

Instance-Config.xml
Conclusión

English Version: The dependency Injector Pattern

Introducción

En el desarrollo de software moderno se ha comenzado a generalizar el uso de patrones para implementar partes de nuestros desarrollos. Los patrones proponen soluciones estándares a los problemas comunes de diseño de nuestras aplicaciones. Hay muchos patrones diferentes, aunque sin duda los más famosos son los conocidos como patrones "GoF".
Realmente podemos definir nosotros mismos otros patrones , siempre que encontremos una solución estándar a un problema común.

En este artículo os voy a mostrar un patrón muy sencillo, el patrón "inyector de dependencias", muy utilizado últimamente en los desarrollos de software modernos, sobre todo en el mundo "Open Source" de Java. Este patrón a su vez deriva de uno más genérico, el patrón de "inversión del control" de Larman.

Como siempre, lo primero que vamos a describir es el problema. El escenario del que vamos a partir es el siguiente:

- Una clase A está implementando una cierta funcionalidad de la aplicación. Como ejemplo podría ser una cierta funcionalidad de la lógica de negocio.
- Para completar dicha funcionalidad usa una clase B de utilidad. Es el caso habitual de las clases que implementan la persistencia de los objetos en la base de datos.
- Para realizar su función, la clase B utiliza otra clase S que le permite el acceso a otros recursos, en nuestro ejemplo será la clase de conexión al gestor de la base de datos.

Diseños tradicionales

Normalmente en los diseños habituales, la clase B de utilidad tiene una relación de uso de la clase S, por lo que para poder utilizarla debe cumplir dos cosas:

- debe conocer su tipo
- debe poder obtener una instancia de la clase S.

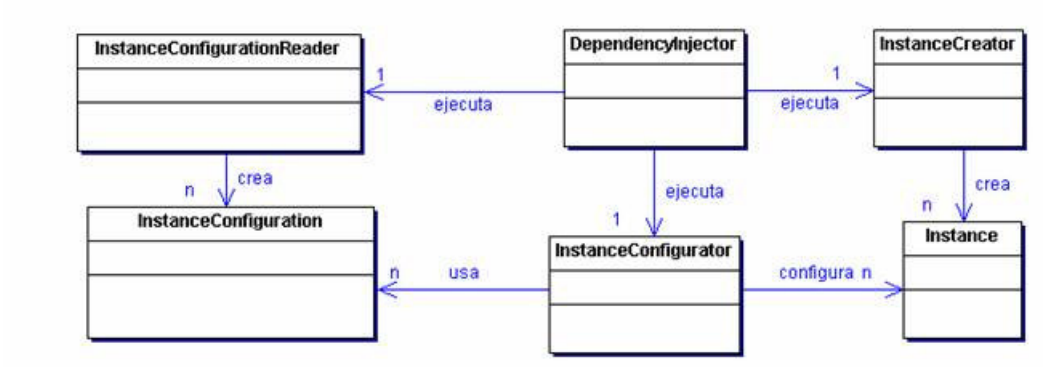
Esto introduce una relación de dependencia, al menos en la dirección B->S, entre ambas clases. Para poder minimizar el impacto de dicha dependencia podemos recurrir a varias soluciones.

- Con respecto al tipo, podemos utilizar el patrón de Interfaz para la clase S, es decir, que B conocerá una interfaz genérica de la clase S, lo que aisla a la clase B de la implementación real de la clase S.
- Ahora nos queda la segunda parte del problema, y es el modo en el que B obtiene una instancia de la clase S. Aquí nos va a ayudar el patrón de inversión de control, que nos dice que la instancia de la clase S le será pasada por la clase A, es decir, los objetos que B necesita para completar la petición de A le son pasados a B como parámetros de la petición. El control sobre el objeto S pasa de la clase llamada B a la clase llamadora A, es decir, se ha invertido el control de la clase S.

Diseñando nuestro patrón de inyección de dependencias DependencyInyector

Podemos crear una versión sencilla de este patrón utilizando un diseño bastante simple.
Este patrón lo vamos a descomponer en los siguientes objetos:

1. **DependencyInyector**. Es el objeto que controla cómo se realiza la inyección de dependencias. Utiliza los objetos DependencyConfigarationReader, InstanceCreator e InstanceConfigurator. La configuración de las instancias la almacena en el mapa de instanceConfugurations y las instancias en el mapa instances.
2. **InstanceConfigurationReader**. Este objeto lee la configuración de dependencias que se va a utilizar. Esta configuración está compuesta de una serie de objetos que se deben instanciar, la lista de parámetros con su valores que se debe proporcionar a cada objeto para su configuración y las relaciones que se deben crear entre los diferentes objetos.
3. **InstanceCreator.** Este es un objeto que sigue el patrón factoría, y se ocupa de crear instancias y ponerlas a disposición de los demás objetos del patrón.
4. **InstanceConfigurator**. Este objeto conoce cómo se debe configurar cada uno de los objetos instanciados, asignándole los parámetros y relaciones.



En el diagrama se puede ver el diseño básico del patrón. La clase principal DependencyInyector sirve como interfaz para el conjunto. El usuario interactuará con el conjunto a través de esta clase. Para diseñar esta clase podemos usar el patrón singleton, ya que normalmente nos hará falta una única instancia de la misma.

Esquema de funcionamiento

Al instanciar un inyector de dependencias, este objeto usa el objeto InstanceConfiguratorReader para que éste obtenga la información de dependencias a utilizar. A continuación se instancia un objeto InstanteCreator para que se puedan crear las instancias de objetos solicitadas. Luego el DependencyInyector crea una instancia del DependencyConfigurator y le pasa a esta instancia las referencias a los otros objetos instanciados (inversión de control en esta instancia, como se puede ver). Ahora es el DependencyConfigurator el responsable de ir creando instancias de objetos, mediante el InstanceCreator, y luego irlas configurando, según las instrucciones que reciba del DependencyConfiguration.

Ejemplo de implementación en Java 5

Para ilustrar el patrón DependencyInjector he creado una implementación simple en Java 5. Esta implementación se puede probar de una manera muy sencilla sin más que crear un proyecto Java en nuestro entorno de desarrollo y añadir los siguientes ficheros:

Init.java

```
public class Init {

	/**
	 * Aplicación simple de prueba del patrón de Inyección de dependencias
	 * EN: Simple application for testing the Dependency Injector Pattern
	 * @param args
	 * @author Cristóbal González Almirón, para  AdictosAlTrabajo.com
	 */
	public static void main(String[] args) {

		DependencyInyector dependencyInjector = new DependencyInyector();

		//Hace el trabajo  ;EN: Do the work
		dependencyInjector.execute();

		//Ver el resultado   ;EN: view result
		InstanceA ia = (InstanceA) dependencyInjector.getInstances().get("instancia1");
		InstanceB ib = (InstanceB) dependencyInjector.getInstances().get("instancia2");

		//EN: show the content of the instances
		System.out.println("Instancia1 =" + ia.toString());
		System.out.println("Instancia2 =" + ib.toString());
		System.out.println("   parama =" + ib.getParama());
		System.out.println("   instancea =" + ib.getInstancea().toString());

		}
}
```

Esta es la clase que crea la instancia del inyector de dependencias. Una vez instanciadas la clase InstanciaA y InstanciaB muestra su contenido. Es un ejemplo simple.

InstanceA.java

```
/**
 * Clase de ejemplo InstanceA
 * EN: samle class InstanceA
 * @author Cristóbal González Almirón
 *
 */
public class InstanceA {
	private String paramx;
	private String paramy;

	public String getParamx() { return paramx; }
	public void setParamx(String paramx) {
		this.paramx = paramx;
	}
	public String getParamy() {  return paramy; }
	public void setParamy(String paramy) {
		this.paramy = paramy;
	}


}
```

InstanceB.jave

```
/**
 * Clase de ejemplo InstanceB
 * EN: samle class InstanceB
 * @author Cristóbal González Almirón
 *
 */
public class InstanceB {
	private String parama;
	private Object instancea;

	public String getParama() {  return parama; }
	public void setParama(String parama) {
		this.parama = parama;
	}
	public Object getInstancea() { return instancea; }
	public void setInstancea(Object instancea) {
		this.instancea = instancea;
	}
}
```

Clases de ejemplo.

DependencyInjector.java

```
import java.util.HashMap;
import java.util.Map;

/**
 * Clase principal del patrón DependencyInjector
 * EN: main class of the DependencyInjector pattern
 * @author Cristóbal González
 *
 */
public class DependencyInyector {
```

```
	private Map<String,Object> instances
			= new HashMap<String, Object>();
	private Map<String,InstanceConfiguration> instanceConfigurations
			= new HashMap<String, InstanceConfiguration>();

	private InstanceConfigurationReader instanceConfigurationReader =
		new InstanceConfigurationReader();

	private InstanceConfigurator instanceConfigurator =
		new InstanceConfigurator();

	private InstanceCreator instanceCreator =
		new InstanceCreator();

	public Map<String, Object> getInstances() {
		return instances;
	}
	public Map<String, InstanceConfiguration> getInstanceConfigurations() {
		return instanceConfigurations;
	}

	/** método principal para ejecutar el patrón
	 * EN: main method to execute the
	 */
	public void execute() {
		instanceConfigurationReader.readConfiguration(instanceConfigurations);
		instanceCreator.createInstances(instanceConfigurations, instances);
		instanceConfigurator.configureInstances(
				instanceConfigurations,
				instances);
	}
}
```

La clase DependencyInjector, responsable de coordinar la inyección de dependencias.

InstanceConfigurationReader.java

```
import java.io.IOException;
import java.util.Map;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

//Clase simple de utilidad para leer la lista de instancias y parámetros
//desde un fichero de configuración en formato XML
//EN: simple utility class to read the `parameters and instances
// configuration list from a xml file
public class InstanceConfigurationReader {


	/** lee la configuraci'on y crea la lista de
	 * instanceConfiguratios
	 */
	public void readConfiguration(
			Map<String, InstanceConfiguration> instanceConfigurations) {
		Document document = null;

		//Carga del fichero de configuración
		//EN: load de xml configuration file
		DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
		DocumentBuilder builder;
		System.out.println("###DependencyConfiguratorReader.readConfiguration");
		try {
			builder = factory.newDocumentBuilder();
			document = builder.parse(
					this.getClass().getResourceAsStream("instance-config.xml"));

			// Creación de las configuraciones
			NodeList list = document.getElementsByTagName("instance");
			for (int i=0; i<list.getLength(); ++i){
				Element element = (Element) list.item(i);

				//creamos la configuración de la instancia
				InstanceConfiguration instanceConfiguration =
						new InstanceConfiguration();
				instanceConfiguration.setName(element.getAttribute("name"));
				instanceConfiguration.setType(element.getAttribute("type"));

				NodeList parameterNodes = element.getElementsByTagName("parameter");
				for (int j=0; j < parameterNodes.getLength(); ++j) {
					Element parameterNode = (Element) parameterNodes.item(j);
					ParameterConfiguration pc = new ParameterConfiguration();
					pc.setName(parameterNode.getAttribute("name"));
					pc.setType(parameterNode.getAttribute("type"));
					pc.setValue(parameterNode.getAttribute("value"));
					instanceConfiguration.getParameters().add(pc);
				}
				instanceConfigurations.put(element.getAttribute("name"),
						instanceConfiguration);
			}
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (ParserConfigurationException e) {
			e.printStackTrace();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Clase simple para leer la información de configuración de las dependencias. Lee la información almacenada en un sencillo fichero XML.

InstanceConfiguration.java

```
import java.util.ArrayList;
import java.util.List;

/**
 * JavaBean que almacena la configuración de una instancia
 * EN: simple JavaBean to hold the instance configuration data
 * @author Cristóbal González Almirón
 *
 */
public class InstanceConfiguration {

        private String name;
        private String type;
        private String value;
        private List<ParameterConfiguration> parameters =
                new ArrayList<ParameterConfiguration>();

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
        public String getType() {
                return type;
        }
        public void setType(String type) {
                this.type = type;
        }
        public String getValue() {
                return value;
        }
        public void setValue(String value) {
                this.value = value;
        }

        public List<ParameterConfiguration> getParameters() {
                return this.parameters;
        }

        public void setParameters(List<ParameterConfiguration> parameters) {
                this.parameters = parameters;
        }
}
```

Java Bean para para amacenar la información de configuración de dependencias.

ParameterConfiguration.java

```
/**
 * JavaBean simple para almacenar los datos de un parámetro de
 * configuración de una instancia
 * EN: simple JavaBean to hold the data of a configuration parameter
 * @author Cristóbal González Almirón
 *
 */
public class ParameterConfiguration {
        private String name;
        private String type;
        private String value;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getType() {
                return type;
        }
        public void setType(String type) {
                this.type = type;
        }
        public String getValue() {
                return value;
        }
        public void setValue(String value) {
                this.value = value;
        }
}
```

Clase auxiliar del bean de configuración.

InstanceCrearor.java

```
import java.util.Map;

/**
 * Factoría simple para crear instancias de objetos
 * EN: simple factory to create Object instances
 */
public class InstanceCreator {


	/** Factoría abstracta genérica.  Crea instancias de clases a partir
	 * de su numbre. La clase debe tener contructor sin parámetros.*/
	public Object getNewInstance(String type){
		try {
			Class clase = Class.forName(type);
			return clase.newInstance();

		} catch (ClassNotFoundException e) {
			System.out.println("Error de instanciación, clase no encontrada:" + type);
			e.printStackTrace();
		} catch (InstantiationException e) {
			System.out.println("Error de instanciación:" + type);
			e.printStackTrace();
		} catch (IllegalAccessException e) {
			System.out.println("Error de instanciación, acceso ilegal:" + type);
			e.printStackTrace();
		}
		return null;
	}

	//Crea todas las instancias definidas en InstanceConfigurations, usando
	//una factoría abstracta simple, y añadiéndolos a la lista de instancias Instances
	//EN: creates the instances defined in the instanceConfiguration list and
	// adds each instance to the map intances.
	public void createInstances(Map<String, InstanceConfiguration> instanceConfigurations,
			Map<String, Object> instances){
		System.out.println("###InstanceCreator.createInstance");
		for (Map.Entry<String, InstanceConfiguration> entry: instanceConfigurations.entrySet()){
			InstanceConfiguration ic = entry.getValue();
			instances.put(ic.getName(), getNewInstance(ic.getType()));
			System.out.println("New instance: " + ic.getName() + ":"+ ic.getType());
		}
	}
}
```

Creador de instancias. Para este ejemplo he usado una factoría abstracta muy simple.

InstanceConfigurator.java

```
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/*^*
 * Configurador de instancias genérico simple
 * Funcionamiento: asigna valores a las diferentes propiedades
 * de una instancia, usando el API de reflexión de Java
 * EN: simle generic intance configurator
 * Configures the properties of an instance of a bean, using the
 * Java reflection API
 */
public class InstanceConfigurator {

	List<ParameterConfiguration> parameters =
		new ArrayList<ParameterConfiguration>();


	public void configureInstances(Map<String, InstanceConfiguration> instanceConfigurations,
			Map<String, Object> instances){

		System.out.println("###instanceConfigurator.configureInstances");
		for (Map.Entry<String, InstanceConfiguration> entry: instanceConfigurations.entrySet()){

			InstanceConfiguration ic = entry.getValue();
			Object instance = instances.get(ic.getName());
			System.out.println(">Configurando Instancia: " + ic.getName() + ": classv" +
ic.getType() + "= (value if String)" + ic.getValue());
			try {

				//para cada parámetro se ajusta su valor
				//EN: set value for each parameter
				for (ParameterConfiguration parameter : ic.getParameters()) {
					System.out.println("Procesando parámetro " + parameter.getName());

					//Si el parámetro es de tipo String asigna el valor a la propiedad
					//EN: if the param is a String, set the value to the property
					if (parameter.getType().equals("String")) {
						Field attribute = instance.getClass().getDeclaredField
(parameter.getName());
						attribute.setAccessible(true);
						//si es un String el valor del parámetro se asigna
directamente
						attribute.set(instance, parameter.getValue());
						System.out.println("  Asignado a " + parameter.getName() +
" el string " + parameter.getValue());

					//Si el parámetro es un objeto, asigna la instancia de la lista de
```

```
instancias
                                    //creada con el InstanceCreator
                                    //EN: if the parameter is an Object, sets the property with an
Objject instance
                                    //created with the InstanceCreator
                                    } else if (parameter.getType().equals("Object")) {
                                          Field objectAttribute = instance.getClass
().getDeclaredField (parameter.getName());
                                          objectAttribute.setAccessible(true);
                                          //Si es un objeto el valor del parámetro es el índice en la
tabla de instancias
                                          objectAttribute.set(instance,instances.get
(parameter.getValue()));
                                          System.out.println("  Asignado a " + parameter.getName() +
" el objeto " + parameter.getValue());

                                    } else
                                          System.out.println("  No es String o Object");

                              }
                        } catch (IllegalArgumentException e) {
                              e.printStackTrace();
                        } catch (SecurityException e) {
                              e.printStackTrace();
                        } catch (IllegalAccessException e) {
                              e.printStackTrace();
                        } catch (NoSuchFieldException e) {
                              e.printStackTrace();
                        }
                  }
            }

      public void addParameter(String name, String type, String value){
            ParameterConfiguration param = new ParameterConfiguration();
            param.setName(name);
            param.setType(type);
            param.setValue(value);
            parameters.add(param);
      }
}
```

Configurador de instancias. Utiliza la información de configuración para inyectar parámetros y dependencias de otras clases. Utiliza el mecanismo de reflexión de Java.

Instance-Config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<instances>
      <instance name="instancia1" type="InstanceA">
            <parameter name="paramx" type="String" value="X" />
            <parameter name="paramy" type="String" value="Y" />
      </instance>
      <instance name="instancia2" type="InstanceB">
            <parameter name="parama" type="String" value="A" />
            <parameter name="instancea" type="Object" value="instancia1" />
      </instance>
</instances>
```

Fichero de configuración de las instancias.

Conclusión

En este artículo hemos visto cómo utilizar el patrón de inyector de dependencias, que nos permite crear instancias de objetos y configurarlas, aislando las dependencias entre los mismos mediante la inyección de las instancias de otros objetos como parámetros de los objetos.
Además hemos visto una implementación simple del patrón, en Java 5, que nos permitirá añadir de manera sencilla un inyector de dependencias a nuestros proyectos. El proceso de instanciación de los diferentes objetos que componen nuestro sistema o aplicación se controlará mediante un simple ficheo xml, lo que nos permitirá modificar la configuración del mismo sin más que utilizar diferentes ficheros xml.
Este patrón nos resolverá de una manera sencilla y elegante el ensamblado de componentes dentro de nuestra aplicación.
Además con este artículo entenderemos mejor el esquema de funcionamiento de los contenedores de objetos actuales, como puede ser Spring Framework y otros.

Recuerda

Autentia te regala la mayoría del conocimiento aquí compartido (Ver todos los tutoriales). Somos expertos en: J2EE, Struts, JSF, C++, OOP, UML, UP, Patrones de diseño ... y muchas otras cosas.

¿Nos vas a tener en cuenta cuando necesites consultoría o formación en tu empresa?, ¿Vas a ser tan generoso con nosotros como lo tratamos de ser con vosotros?

Somos pocos, somos buenos, estamos motivados y nos gusta lo que hacemos ...

Autentia = Soporte a Desarrollo & Formación.

info@autentia.com

Servicio de notificaciones:

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales.

Formulario de subcripción a novedades:

E-mail Aceptar

Otros Tutoriales Recomendados.

Nombre corto	Descripción
Manual básico de Spring WebFlow	En este tutorial Javier Antoniucci nos enseña cómo empezar a trabajar cpn el framework de desarrollo web Spring webflow.
Patrones de diseño J2EE	Os mostramos una interpretación particular de los patrones de diseño J2EE
Comparativa entre EJB3 y Spring	En este tutorial os mostramos una comparativa entre EJB3 y Spring esperando que os ayude a decidir qué tecnología utilizar.
Patrones de GRASP	Os presentamos una introducción a los patrones de asignación de responsabilides y su relación con el proceso unificado.
Spring WebFlow con Validator	En este tutorial se muestra como podemos realizar las validaciones más frecuentes de datos mediante Spring WebFlow.
Spring WebFlow Tiles	En este tutorial aprenderemos el uso de tiles para usarlo en conjunción con Spring WebFlow.
Aplicación de Patrones de Diseño en Java	En este tutorial os mostramos como las técnicas avanzadas de diseño (como patrones de diseño) contribuyen a la contrucción de aplicaciones profesionales en Java.
Introducción a Spring Web Flow	Spring Web Flow es un módulo de extensión del framework Spring, que facilita la implementación del flujo de páginas de una aplicación web
Spring: definición dinámica de Beans	Este tutorial habla sobre la modificación dinámica de los beans del contexto para simplificar la configuración de Spring
URLs amigables con Spring MVC	En este tutorial se va a hacer un ejemplo práctico utilizando Spring MVC para la configuración de URLs amigables de nuestra aplicación

Ver todos los tutoriales

Nota:

Los tutoriales mostrados en este Web tienen como objetivo la difusión del conocimiento.

Los contenidos y comentarios de los tutoriales son responsabilidad de sus respectivos autores.

En algún caso se puede hacer referencia a marcas o nombres cuya propiedad y derechos es de sus respectivos dueños. Si algún afectado desea que incorporemos alguna reseña específica, no tiene más que solicitarlo.

Si alguien encuentra algún problema con la información publicada en este Web, rogamos que informe al administrador `rcanales@adictosaltrabajo.com` para su resolución.